Operator Overloading and Type Conversions

7

Key Concepts

Overloading | Operator functions | Overloading unary operators | String manipulations | Basic to class type | Class to class type | Operator overloading | Overloading binary operators | Using friends for overloading | Type conversions | Class to basic type | Overloading rules

7.1 Introduction

Operator overloading is one of the many exciting features of C++ language. It is an important technique that has enhanced the power of extensibility of C++. We have stated more than once that C++ tries to make the user-defined data types behave in much the same way as the built-in types. For instance, C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can overload (give additional meaning to) all the C++ operators except the following:

- Class member access operators (., .*).
- Scope resolution operator (::).
- Size operator (sizeof).
- Conditional operator (?:).

The reason why we cannot overload these operators may be attributed to the fact that these operators take names (example class

name) as their operand instead of values, as is the case with other normal operators.

Although the semantics of an operator can be extended, we cannot change its syntax, the grammatical rules that govern its use such as the number of operands, precedence and associativity. For example, the multiplication operator will enjoy higher precedence than the addition operator. Remember, when an operator is overloaded, its original meaning is not lost. For instance, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

7.2 Defining Operator Overloading

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called operator function, which describes the task. The general form of an operator function is:

```
return type classname :: operator op(arglist)
{
        Function body // task defined
}
```

where return type is the type of value returned by the specified operation and op is the operator being overloaded. operator op is the function name, where operator is a keyword.

Operator functions must be either member functions (nonstatic) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the

case with friend functions. Arguments may be passed either by value or by reference. Operator functions are declared in the class using prototypes as follows:

```
vector operator+(vector); //vector addition
vector operator-(); //unary minus
friend vector operator+(vector,vector);  //vector addition
friend vector operator-(vector);  //unary minus
vector operator-(vector &a); // subtraction
int operator==(vector); //comparison
friend int operator==(vector,vector)//comparison
```

vector is a data type of class and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements (as in mathematics).

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function operator op() in the public part of the class. It may be either a member function or a friend function.
3. Define the operator function to implement the required operations.
4. Overloaded operator functions can be invoked by expressions such as

   op x or x *op*

for unary operators and

x op y

for binary operators. op x (or x op) would be interpreted as

operator *op (x)*

for friend functions. Similarly, the expression x op y would be interpreted as either

x.operator *op (y)*

in case of member functions, or

operator *op (x,y)*

in case of friend functions. When both the forms are declared, standard argument matching is applied to resolve any ambiguity.

7.3 Overloading Unary Operators

Let us consider the unary minus operator. A minus operator when used as a unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an int or float variable. The unary minus when applied to an object should change the sign of each of its data items.

Program 7.1 shows how the unary minus operator is overloaded.

Program 7.1 Overloading Unary Minus

```
#include <iostream>
using namespace std;
class space
{
int x;
int y;
int z;
```

```
public:
void getdata(int a, int b, int c);
void display(void);
void operator-();// overload unary minus
};
void space :: getdata(int a, int b, int c)
{
x = a;
y = b;
z = c;
}
void space :: display(void)
{
cout<< "x = "<<x<<" ";
cout<<"y = "<<y<<" ";
cout<<"z = "<<z<<"\n";
}
void space :: operator-()
{
x = -x;
y = -y;
z = -z;
}
int main()
{
space S;
S.getdata(10, -2 0, 30); cout << "S : "; S.display();
-S;// activates operator-() function
cout << "-S : ";
S.display();
return 0;
}
```

The output of Program 7.1 would be:

S : x = 10 y = -20 z = 30

-S : x = -10 y = 20 z = -30

NOTE: The function operator - () takes no argument. Then, what does this operator function do? It changes the sign of data members of the object S. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

Remember, a statement like

```
S2 = -S1;
```

will not work because, the function operator-() does not return any value. It can work if the function is modified to return an object.

It is possible to overload a unary minus operator using a friend function as follows:

```
friend void operator-(space &s);// declaration
void operator-(space &s)// definition
{
s.x = -s.x;
s.y = -s.y;
s.z = -s.z;
}
```

NOTE: *Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator-() Therefore, the changes made inside the operator function will not reflect in the called object.*

7.4 Overloading Binary Operators

We have just seen how to overload an unary operator. The same mechanism can be used to overload a binary operator. In Chapter 6, we illustrated, how to add two complex numbers using a friend function. A statement like

```
C = sum(A, B); // functional notation.
```

was used. The functional notation can be replaced by a natural looking expression

```
C = A + B;  // arithmetic notation
```

by overloading the + operator using an operator+() function. The Program 7.2 illustrates how this is accomplished.

Program 7.2 Overloading + Operator

```
#include <iostream>
using namespace std;
class complex
{
float x;  //real part
float y; // imaginary part
public:
complex(){ } //constructor 1
complex(float real, float imag) //constructor 2
{ x = real; y = imag; }
complex operator+(complex); void display(void);
};
complex complex :: operator+(complex c)
{
```

```
        complex temp; //temporary
        temp.x = x + c.x; // these are
        temp.y = y + c.y; //float additions
        return(temp);
        }
        void complex :: display(void)
{
        cout << x << " + j" << y << "\n";
        int main()
{
        complex C1, C2, C3; //invokes constructor 1
        C1 = complex(2.5, 3.5); // invokes constructor 2
        C2 = complex(1.6, 2.7);
        C3 = C1 + C2;
        cout << "C1 = "; C1.display();
        cout << "C2 = "; C2.display();
        cout << "C3 = "; C3.display();
        return 0;
        }
```

The output of Program 7.2 would be:

C1 = 2.5 + j3.5

C2 = 1.6 + j2.7

C3 = 4.1 + j6.2

NOTE: *Let us have a close look at the function operator+() and see how the operator overloading is implemented*.

```
complex complex :: operator+(complex c)
{
complex temp;
temp.x = x + c.x;
```

```
temp.y = y + c.y;
return(temp);
}
```

We should note the following features of this function:

1. It receives only one complex type argument explicitly.
2. It returns a complex type value.
3. It is a member function of complex.

The function is expected to add two complex values and return a complex value as the result but receives only one value as argument. Where does the other value come from? Now let us look at the statement that invokes this function:

```
C3 = C1 + C2; //  invokes operator+() function
```

We know that a member function can be invoked only by an object of the same class. Here, the object C1 takes the responsibility of invoking the function and C2 plays the role of an argument that is passed to the function. The above invocation statement is equivalent to

```
C3 = C1.operator+(C2); //  usual function call syntax
```

Therefore, in the operator+() function, the data members of C1 are accessed directly and the data members of C2 (that is passed as an argument) are accessed using the dot operator. Thus, both the objects are available for the function. For example, in the statement

```
temp.x = x + c.x;
```

c.x refers to the object C2 and x refers to the object C1.temp.x is the real part of temp that has been created specially to hold the results of addition of C1 and C2. The function returns the complex temp to be assigned to C3. Figure 7.1 shows how this is implemented.

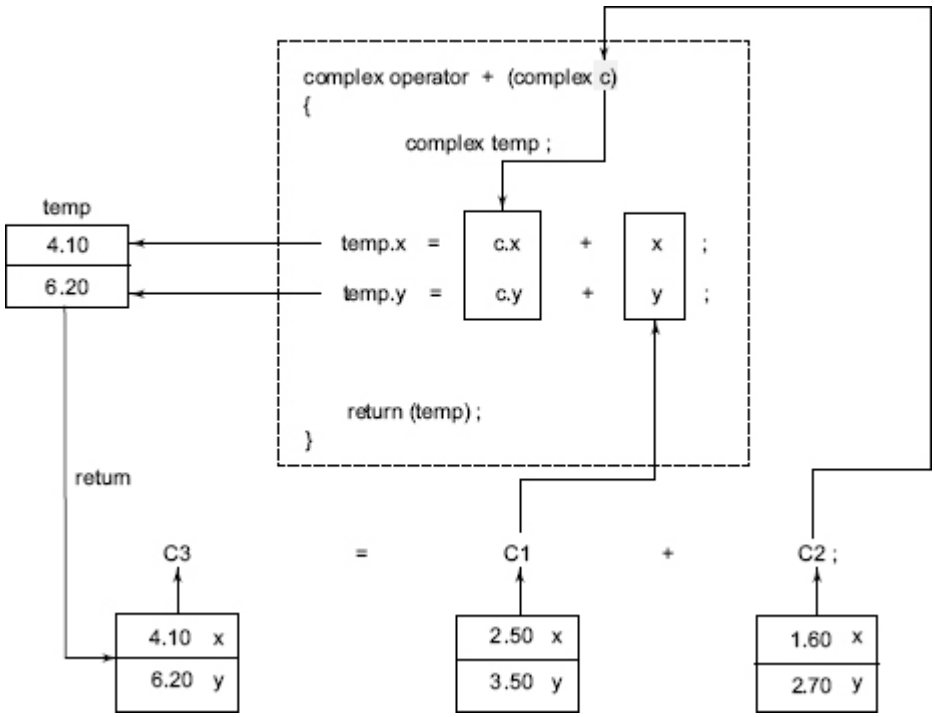

Fig. 7.1 *Implementation of the overloded + operator*

As a rule, in overloading of binary operators, the left-hand operand is used to invoke the operator function and the right-hand operand is passed as an argument.

We can avoid the creation of the temp object by replacing the entire function body by the following statement:

```
return complex((x+c.x),(y+c.y)); // invokes constructor 2
```

What does it mean when we use a class name with an argument list? When the compiler comes across a statement like this, it invokes an appropriate constructor, initializes an object with no name and returns the contents for copying into an object. Such an object is called a temporary object and goes out of space as soon as the contents are assigned to another object. Using temporary objects can make the code shorter, more efficient and better to read.

7.5 Overloading Binary Operators Using Friends

As stated earlier, friend functions may be used in the place of member functions for overloading a binary operator, the only difference being that a friend function requires two arguments to be explicitly passed to it, while a member function requires only one.

The complex number program discussed in the previous section can be modified using a friend operator function as follows:

1. Replace the member function declaration by the friend function declaration.

```
friend complex operator+(complex, complex);
```

2. Redefine the operator function as follows:

```
complex operator+(complex a, complex b)
{
return complex((a.x+b.x),(a.y+b.y));
}
```

In this case, the statement

```
C3 = C1 + C2;
```

is equivalent to

```
C3 = operator+(C1, C2);
```

In most cases, we will get the same results by the use of either a friend function or a member function. Why then an alternative is made available? There are certain situations where we would like to use a friend function rather than a member function. For instance, consider a situation where we need to use two different types of operands for a binary operator, say, one an object and another a built-in type data as shown below,

```
A = B + 2; (or A = B * 2;)
```

where A and B are objects of the same class. This will work for a member function but the statement

A = 2 + B; (or A = 2 * B)

will not work. This is because the left-hand operand which is responsible for invoking the member function should be an object of the same class. However, friend function allows both approaches. How?

It may be recalled that an object need not be used to invoke a friend function but can be passed as an argument. Thus, we can use a friend function with a built-in type data as the left-hand operand and an object as the right-hand operand. Program 7.3 illustrates this, using scalar multiplication of a vector. It also shows how to overload the input and output operators >> and <<.

Program 7.3 Overloading Operators Using Friends

```
#include <iostream.h>
const size = 3;
class vector
{
int v[size];
public:
vector(); //  constructs null vector
vector(int *x); //  constructs vector from array
friend vector operator *(int a, vector b); //  friend 1
friend vector operator *(vector b, int a); //  friend 2
friend istream & operator >> (istream &, vector &);
friend ostream & operator << (ostream &, vector &);
};
vector :: vector()
```

```
{
for(int i=0; i<size;i++)
v[i] = 0;
}
vector :: vector(int *x)
{
for(int i=0; i<size; i++)
v[i] = x[i];
}
vector operator *(int a, vector b)
{
vector c;
for(int i=0; i < size; i++)
c.v[i] = a * b.v[i];       return c;
}
vector operator *(vector b, int a)
{
vector c;
for(int i=0; i<size;
c.v[i] = b.v[i] * a;
return c;
}
istream & operator >> (istream &din, vector &b)
{
for(int i=0; i<size; i++)
din >> b.v[i];
return(din);
}
ostream & operator << (ostream &dout, vector &b)
{
dout << "(" << b.v [0];
for(int i=1; i<size;
dout << ", " << b.v[i];
dout << ")";
return(dout);
}
int x[size] = {2,4,6};
```

```
int main()
{
vector m; // invokes constructor 1
vector n = x; // invokes constructor 2
cout << "Enter elements of vector m " << "\n";
cin >> m; // invokes operator>>() function
cout << "\n";
cout << "m = " << m << "\n"; // invokes operator <<()
vector p, q;
p = 2 * m; // invokes friend 1
q = n * 2; // invokes friend 2
cout << "\n";
cout << "p = " << p << "\n"; // invokes operator<<()
cout << "q = " << q << "\n";
return 0;
}
```

The output of Program 7.3 would be:

```
Enter elements of vector m
5 10 15
m = (5, 10, 15)
p = (10, 20, 30)
q = (4, 8, 12)
```

The program overloads the operator * two times, thus overloading the operator function opera-tor*() itself. In both the cases, the functions are explicitly passed two arguments and they are invoked like any other overloaded function, based on the types of its arguments. This enables us to use both the forms of scalar multiplication such as

```
p = 2 * m; // equivalent to p = operator*(2,m);
q = n * 2; // equivalent to q = operator*(n,2);
```

The program and its output are largely self-explanatory. The first constructor

```
vector();
```

constructs a vector whose elements are all zero. Thus

```
vector m;
```

creates a vector m and initializes all its elements to 0. The second constructor

```
vector(int &x);
```

creates a vector and copies the elements pointed to by the pointer argument x into it. Therefore, the statements

```
int x[3] = {2, 4, 6};
vector n = x;
```

create n as a vector with components 2, 4, and 6.

NOTE: *We have used vector variables like m and n in input and output statements just like simple variables. This has been made possible by overloading the operators >> and << using the functions:*

```
friend istream & operator>> (istream &, vector &);
friend ostream & operator<<(ostream &, vector &);
```

istream and ostream are classes defined in the iostream.h file which has been included in the program.

7.6 Manipulation of Strings Using Operators

ANSI C implements strings using character arrays, pointers and string functions. There are no operators for manipulating the strings.

One of the main drawbacks of string manipulations in C is that whenever a string is to be copied, the programmer must first determine its length and allocate the required amount of memory.

Although these limitations exist in C++ as well, it permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers. (Recently, ANSI C++ committee has added a new class called string to the C++ class library that supports all kinds of string manipulations. String manipulations using the string class are discussed in Chapter 15.

For example, we shall be able to use statements like

```
string3 = string1 + string2;
if(string1 >= string2) string = string1;
```

Strings can be defined as class objects which can be then manipulated like the built-in types. Since the strings vary greatly in size, we use new to allocate memory for each string and a pointer variable to point to the string array. Thus we must create string objects that can hold these two pieces of information, namely, length and location which are necessary for string manipulations. A typical string class will look as follows:

```
class string
{
char *p; //  pointer to string
int len; //  length of string
public:
....  //  member functions
....  //  to initialize and
....  //  manipulate strings
};
```

We shall consider an example to illustrate the application of overloaded operators to strings. The example shown in Program 7.4

overloads two operators, + and <= just to show how they are implemented. This can be extended to cover other operators as well.

Program 7.4 Mathematical Operations on Strings

```
#include <string.h>
#include <iostream.h>
class string
{
char *p;
int len;
public:
string() {len = 0; p = 0;}//  create null string
string(const char * s); //  create string from arrays
string(const string & s); //  copy constructor
~ string(){delete p;}//  destructor
// + operator
friend string operator+(const string &s, const string &t);
// <= operator
friend int operator<=(const string &s, const string &t);
friend void show(const string s);
};
string :: string(const char *s)
{
len = strlen(s);
p = new char[len+1];
strcpy(p,s);
}
string :: string(const string & s)
{
len = s.len;
p = new char[len+1];
strcpy(p,s.p);
```

```
}
// overloading + operator
string operator+(const string &s, const string &t)
{
string temp;
temp.len = s.len + t.len;
temp.p = new char[temp.len+1];
strcpy(temp.p,s.p);
strcat(temp.p,t.p);
return(temp);
// overloading <= operator
int operator<=(const string &s, const string &t)
{
int m = strlen(s.p);
int n = strlen(t.p);
if(m <= n) return(1);
else return(0);
}
void show(const string s)
{
cout << s.p;
}
int main()
{
string s1 = "New ";
string s2 = "York";
string s3 = "Delhi";
string string1, string2, string3;
string1 = s1;
string2 = s2;
string3 = s1+s3;
cout << "\nstring1 = "; show(string1);
cout << "\nstring2 = "; show(string2);
cout <<  "\n";
cout <<  "\nstring3 = "; show(string3);
cout <<  "\n\n";
if(string1 <= string3)
```

```
{
show(string1);
cout << " smaller than ";
show(string3);
cout << "\n";
}
else
{
show(string3);
cout << " smaller than ";
show(string1);
cout << "\n";
}
return 0;
}
```

The output of Program 7.4 would be:

```
string1 = New
string2 = York
string3 = New Delhi
New smaller than New Delhi
```

7.7 Some Other Operator Overloading Examples

Overloading the Subscript Operator []

The subscript operator is normally used to access and modify a specific element in an array. Program 7.5 demonstrates the overloading of the subscript operator to customize its behaviour:

Program 7.5 Overloading of the Subscript Operator

```
#include<iostream&62;
#include<conio.h>
using namespace std;
class arr
{
int a[5];
public:
arr(int *s)
{
int i;
for(i=0;i<5;i++)
a[i]=s[i];
}
int operator [] (int k) //  Overloading the subscript operator
{
return (a[k]);
}
};
int main()
{
int x[5] = {1,2,3,4,5};
arr A(x); int i;
for(i=0;i<5;i++)
{
cout<<x[i]<<"\t"; //  Using subscript operator to access the
private array elements
}
getch();
return 0;
}
```

The output of Program 7.5 would be:

1 2 3 4 5

As can be seen in the above program, we have used the subscript operator along with the object name to access the private array elements of the object.

Overloading the Pointer-to-member (->) Operator

The pointer-to-member operator (->) is normally used in conjunction with an object pointer to access any of the object's members. Program 7.6 demonstrates the overloading of the -> operator:

Program 7.6 Overloading of Pointer-to-member Operator

```
#include<iostream>
#include<conio.h>
using namespace std;
class test
{
public:
int num;
test(int j)
{
num=j;
}
test* operator ->(void)
{
return this;
}
};
int main()
{
test T(5);
test *Ptr = &T;
cout<<"T.num = "<<T.num;  // Accessing num normally
```

```
    cout<<"\nPtr->num = "<<Ptr->num;  // Accessing num using
normal object pointer
    cout<<"\nT->num = "<<T->num;  // Accessing num using
opverloaded -> operator
    getch();
    return 0;
    }
```

The output of Program 7.6 would be:

```
T.num = 5
Ptr->num = 5
T->num = 5
```

The above program demonstrates both the normal (Ptr->num) as well as the overloaded (T->num) behaviour of the -> operator.

The statement,

```
return this;
```

returns a pointer to itself; that is a pointer to the test class object.

7.8 Rules for Overloading Operators

Although it looks simple to redefine the operators, there are certain restrictions and limitations in overloading them. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
4. There are some operators that cannot be overloaded. (See Table 7.1.)

5. We cannot use friend functions to overload certain operators. (See Table 7.2.) However, member functions can be used to overload them.
6. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
7. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
8. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
9. Binary arithmetic operators such as +, -, *, and / must explicitly return a value. They must not attempt to change their own arguments.

7.9 Type Conversions

We know that when constants and variables of different types are mixed in an expression, C applies automatic type conversion to the operands as per certain rules. Similarly, an assignment operation also causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of the variable on the left. For example, the statements

```
int m;
float x = 3.14159;
m = x;
```

convert x to an integer before its value is assigned to m. Thus, the fractional part is truncated. The type conversions are automatic as long as the data types involved are built-in types.

What happens when they are user-defined data types?

Consider the following statement that adds two objects and then assigns the result to a third object.

```
v3 = v1 + v2; // v1, v2 and v3 are class type objects
```

When the objects are of the same class type, the operations of addition and assignment are carried out smoothly and the compiler does not make any complaints. We have seen, in the case of class objects, that the values of all the data members of the right-hand object are simply copied into the corresponding members of the object on the left-hand. What if one of the operands is an object and the other is a built-in type variable? Or, what if they belong to two different classes?

Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types. We must, therefore, design the conversion routines by ourselves, if such operations are required.

Three types of situations might arise in the data conversion between uncompatible types:

1. Conversion from basic type to class type.
2. Conversion from class type to basic type.
3. Conversion from one class type to another class type.

We shall discuss all the three cases in detail.

Basic to Class Type

The conversion from basic type to class type is easy to accomplish. It may be recalled that the use of constructors was illustrated in a number of examples to initialize objects. For example, a constructor was used to build a vector object from an int type array. Similarly, we used another constructor to build a string type object from a char* type variable. These are all examples where constructors perform a defacto type conversion from the argument's type to the constructor's class type.

Consider the following constructor:

```
string :: string(char *a)
{
length = strlen(a);
P = new char[length+1];
strcpy(P,a);
}
```

This constructor builds a string type object from a char* type variable a. The variables length and p are data members of the class string. Once this constructor has been defined in the string class, it can be used for conversion from char* type to string type. Example:

```
string s1, s2;
char* name1 = "IBM PC";
char* name2 = "Apple Computers";
s1 = string(name1);
s2 = name2;
```

The statement

```
s1 = string(name1);
```

first converts name1 from char* type to string type and then assigns the string type values to the object s1. The statement

```
s2 = name2;
```

also does the same job by invoking the constructor implicitly.

Let us consider another example of converting an int type to a class type.

```
class time
{
int hrs;
int mins;
```

```
public:
.....
.....
time(int t) //constructor
{
hrs = t/60; // t in minutes
mins = t%60;
}
};
```

The following conversion statements can be used in a function:

```
time T1; //object T1 created
int duration = 85;
T1 = duration; // int to class type
```

After this conversion, the hrs member of T1 will contain a value of 1 and mins member a value of 25, denoting 1 hours and 25 minutes.

NOTE: *The constructors used for the type conversion take a single argument whose type is to be converted*.

In both the examples, the left-hand operand of = operator is always a class object. Therefore, we can also accomplish this conversion using an overloaded = operator.

Class to Basic Type

The constructors did a fine job in type conversion from a basic to class type. What about the conversion from a class to basic type? The constructor functions do not support this operation. Luckily, C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type. The general form of an overloaded casting operator function, usually referred to as a conversion function, is:

```
Operator typename()
{
    ....
    ..... (Function statements)
    .....
}
```

This function converts a class type data to typename. For example, the operator double() converts a class object to type double, the operator int() converts a class type object to type int, and so on.

Consider the following conversion function:

```
vector :: operator double()
{
double sum = 0;
for(int i=0; i<size; i++
sum = sum + v[i] * v[i];
return sqrt(sum);
}
```

This function converts a vector to the corresponding scalar magnitude. Recall that the magnitude of a vector is given by the square root of the sum of the squares of its components. The operator double() can be used as follows:

```
double length = double(V1);
or
double length = V1;
```

where V1 is an object of type vector. Both the statements have exactly the same effect. When the compiler encounters a statement that requires the conversion of a class type to a basic type, it quietly calls the casting operator function to do the job.

The casting operator function should satisfy the following conditions:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

Since it is a member function, it is invoked by the object and, therefore, the values used for conversion inside the function belong to the object that invoked the function. This means that the function does not need an argument.

In the string example described in the previous section, we can do the conversion from string to char* as follows:

```
string :: operator char*()
{
return(p);
}
```

One Class to Another Class Type

We have just seen data conversion techniques from a basic to class type and a class to basic type. But there are situations where we would like to convert one class type data to another class type.
Example:

```
objX = objY; //objects of different types
```

objX is an object of class **X** and **objY** is an object of class **Y**. The class **Y** type data is converted to the **class X** type data and the converted value is assigned to the **objX**. Since the conversion takes place from **class Y to class X**, **Y** is known as the source class and **X** is known as the destination class.

Such conversions between objects of different classes can be carried out by either a constructor or a conversion function. The compiler treats them the same way. Then, how do we decide which

form to use? It depends upon where we want the type-conversion function to be located in the source class or in the destination class.

We know that the casting operator function

```
Operator typename()
```

converts the class object of which it is a member to typename. The typename may be a built-in type or a user-defined one (another class type). In the case of conversions between objects, typename refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used (i.e., source class). The conversion takes place in the source class and the result is given to the destination class object.

Now consider a single-argument constructor function which serves as an instruction for converting the argument's type to the class type of which it is a member. This implies that the argument belongs to the source class and is passed to the destination class for conversion. This makes it necessary that the conversion constructor be placed in the destination class. Figure 7.2 illustrates these two approaches.

Fig. 7.2 *Conversion between object*

Table 7.3 provides a summary of all the three conversions. It shows that the conversion from a class to any other type (or any other class) should make use of a casting operator in the source class. Or the other hand, to perform the conversion from any other type/class to a class type, a constructor should be used in the destination class.

When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument. Since data members of the source class are private, we must use special access functions in the source class to facilitate its data flow to the destination class.

A Data Conversion Example

Let us consider an example of an inventory of products in store. One way of recording the details of the products is to record their code number, total items in the stock and the cost of each item. Another approach is to just specify the item code and the value of the item in the stock. The example shown in Program 7.7 uses two classes and shows how to convert data of one type to another.

Program 7.7 Data Conversions

```
#include <iostream>
using namespace std;
class invent2; //destination class declared
class invent1 //source class
{
int code; //item code
int items; //no. of items
float price; //cost of each item
public:
invent1(int a, int b, float c)
{
code = a;
items = b;
price = c;
}
void putdata()
{
cout << "Code: " << code << "\n";
cout << "Items: " << items << "\n";
cout << "Value: " << price << "\n";
}
int getcode() {return code;}
int getitems() {return items;}
```

```
float getprice() {return price;}
operator float() {return(items * price);}
/* operator invent2() // invent1 to invent2
{
invent2 temp;
temp.code = code;
temp.value = price * items;
return temp;
 */
}; // End of source class
class invent2 //destination class
{
int code;
float value;
 public:
invent2()
code = 0; value = 0;
}
invent2(int x, float y) // constructor for initialization
code = x;
value = y;
void putdata()
cout << "Code: " << code << "\n";
cout << "Value: " << value << "\n\n";
invent2(invent1 p) //constructor for conversion
code = p.getcode();
value =p.getitems() * p.getprice();
; //End of destination class
int main()
{
invent1 s1(100,5,140.0);
invent2 d1;
float total_value;
/* invent1 To float */
total_value = s1;
/* invent1 To invent2 */
d1 = s1;
```

```
    cout << "Product details - invent1 type" << "\n"; s1.putdata();
    cout << "\nStock value" << "\n";
    cout << "Value = " << total_value << "\n\n";
    cout << "Product details-invent2 type" << "\n";
    d1.putdata();
    return 0;
    }
```

The output of Program 7.7 would be:

```
Product details-invent1 type
Code: 100
Items: 5
Value: 140
Stock value Value = 700
Product details-invent2 type
Code: 100
Value: 700
```

NOTE: *We have used the conversion function*

```
operator float( )
```

in the class ***invent1*** *to convert the* ***invent1*** *type data to a* ***float****. The constructor*

```
invent2 (invent1)
```

is used in the class ***invent2*** *to convert the* ***invent1*** *type data to the* ***invent2*** *type data.*

Remember that we can also use the casting operator function

```
operator invent2()
```

in the class invent1 to convert ***invent1*** *type to* ***invent2*** *type. However, it is important that we do not use both the constructor*

and the casting operator for the same type conversion, since this introduces an ambiguity as to how the conversion should be performed.

Summary

- Operator overloading is one of the important features of C++ language that enhances its exhaustibility.
- Using overloading feature we can add two user defined data types such as objects, with the same syntax, just as basic data types.
- We can overload almost all the C++ operators except the following:
  - class member access operators(., .*)
  - scope resolution operator (::)
  - size operator(sizeof)
  - conditional operator(?:)
- Operator overloading is done with the help of a special function, called operator function, which describes the special task to an operator.
- There are certain restrictions and limitations in overloading operators. Operator functions must either be member functions (nonstatic) or friend functions. The overloading operator must have at least one operand that is of user-defined type.

- The compiler does not support automatic type conversions for the user defined data types. We can use casting operator functions to achieve this.
- The casting operator function should satisfy the following conditions:
  - It must be a class member.
  - It must not specify a return type.
  - It must not have any arguments.

Key Terms

arithmetic notation | binary operators | casting | casting operator | constructor | conversion function | destination class | friend | friend function | functional notation | manipulating strings | operator | operator function | operator overloading | scalar multiplication | semantics | sizeof | source class | syntax | temporary object | type conversion | unary operators

Review Questions

7.1 What is operator overloading?

7.2 Why is it necessary to overload an operator?

7.3 Name the operators that can not be overloaded in C++.

7.4 What is an operator function? Describe the syntax of an operator function.

7.5 How many arguments are required in the definition of an overloaded unary operator?

7.6 A class alpha has a constructor as follows: alpha(int a, double b);

Can we use this constructor to convert types?

7.7 A friend function cannot be used to overload the assignment operator =. Explain why?

7.8 When is a friend function compulsory? Give an example.

7.9 We have two classes X and Y. If a is an object of X and b is an object of Y and we want to say a = b; What type of conversion routine should be used and where?

7.10 State whether the following statements are TRUE or FALSE.

(a) Using the operator overloading concept, we can change the meaning of an operator.

(b) Operator overloading works when applied to class objects only.

(c) Friend functions cannot be used to overload operators.

(d) When using an overloaded binary operator, the left operand is implicitly passed to the member function.

(e) The overloaded operator must have at least one operand that is user-defined type.

(f) Operator functions never return a value.

(g) Through operator overloading, a class type data can be converted to a basic type data.

(h) A constructor can be used to convert a basic type to a class type data.

Debugging Exercises

7.1 Identify the error in the following program.

```
#include <iostream.h>
class Space
{
int mCount;
public:
Space()
{
mCount = 0;
}
Space operator ++()
{
mCount++;
return Space(mCount);
}
};
void main()
{
Space objSpace;
objSpace++;
}
```

7.2 Identify the error in the following program.

```
#include <iostream.h>
enum WeekDays
{
mSunday,
mMonday,
mTuesday,
mWednesday,
mThursday,
mFriday,
```

```
mSaturday
};
bool op==(WeekDays& w1, WeekDays& w2)
{
if(w1== mSunday && w2 == mSunday)
return 1;
else if(w1== mSunday && w2 == mSunday)
return 1;
else if(w1== mSunday && w2 == mSunday)
return 1;
else if(w1== mSunday && w2 == mSunday)
return 1;
else if(w1== mSunday && w2 == mSunday)
return 1;
else if(w1== mSunday && w2 == mSunday)
return 1;
else if(w1== mSunday && w2 == mSunday)
return 1;
else
return 0;
}
void main() {
WeekDays w1 = mSunday, w2 = mSunday;
if(w1==w2)
cout << "Same day";
else
cout << "Different day";
}
```

7.3 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
float mWidth;
float mLength;
public:
```

```
Room()
{
}
Room(float w, float h) :mWidth(w), mLength(h)
{
}
operator float()
{
return (float)mWidth * mLength;
}
float getWidth()
{
}
float getLength()
{
return mLength;
}
};
void main()
{
Room objRoom1(2.5, 2.5);
float fTotalArea;
fTotalArea = objRoom1;
cout << fTotalArea;
}<
```

Programming Exercises

NOTE: For all the exercises that follow, build a demonstration program to test your code.

7.1 Create a class FLOAT that contains one float data member. Overload all the four arithmetic operators so that they operate on the objects of FLOAT. 

7.2 Design a class Polar which describes a point in the plane using polar coordinates radius and angle. A point in polar coordinates is shown in Fig. 7.3.

Use the overloaded + operator to add two objects of Polar.

Note that we cannot add polar values of two points directly. This requires first the conversion of points into rectangular co-ordinates, then adding the corresponding rectangular co-ordinates and finally converting the result back into polar co-ordinates. You need to use the following trigonometric formulae:

```
x = r * cos(a);
y = r * sin(a);
a = atan(y/x); // arc tangent
r = sqrt(x*x + y*y) ;
```

7.3 Create a class MAT of size m x n. Define all possible matrix operations for MAT type objects.

Fig. 7.3 *Polor coordinates of a point*

7.4 Define a class String. Use overloaded == operator to compare two strings.

7.5 Define two classes Polar and Rectangle to represent points in the polar and rectangle systems. Use conversion routines to convert from one system to the other.